CS 470/570 Fall 2017
CS 470/570

• CS 470/570 General Information
• Course Overview
Main Programming Language Topics

• Parallel programming in Java
• OpenMP
• Pthreads
• MPI
• GPU programming
General Issues/Terminology

- Identify possible parallelism
- Match amount of parallelism to the architecture
- Synchronization
- Choose a parallel programming model
- Performance
- Data distribution and IO
General Issues/Terminology

• Parallelism in computer systems
  – Multiprogrammed OS
  – Instruction level parallelism (pipelines)
  – Vector processing
  – Multiprocessor machines
    • Multicores
  – Networks of processors
  – GPUs
Parallel Architectures

• Classics Taxonomy (Flynn)
  – Single Instruction Single Data Stream: SISD
  – Single Instruction Multiple Data Streams: SIMD
  – Multiple Instructions Multiple Data Streams: MIMD
SIMD

- Single control Unit
- Multiple ALUs
- All ALUs execute the same instruction on local data
MIMD

• Multiple control units and ALUs
• Separate stream of control on each processor
• Possible organization
  – Shared Physical Address Space
    • Uniform or Non-uniform Memory Access
    • Cache Coherance issues
  – Distributed Physical Address Space
    • Network of Computers
Interconnection Networks

- Bus
- Crossbar
- Multistage networks
Parallel Programming Models

• How is parallelism expressed?
  – Process
  – Task
  – Thread
  – Implicitly

• How is information accurately shared among parallel actions?

• How are parallel actions synchronized?
Parallel Programming Models

- Shared Address Space Programming
  - Shared variables
- Message Based Programming
  - Send/receive values among cooperating processes
- Programming models are independent of the underlying architecture
- SPMD
  - Single Program Multiple Data
Shared Address Space Programming

• Processes
• Threads
• Directive Model
Unix Processes

• Heavyweight processes
  – Requires the creation of a copy of the parent’s data space
  – Process creation has high overhead

• Example functions
  – fork
  – waitpid
Threads

- Lightweight process
  - Does not require the creation of a copy of the parent’s data space

- Example pthread functions
  - pthread_create
  - pthread_join
Directive Based Parallel Programming

- Directives in source code tell compiler where parallelism should be used
- Threads are implicitly created
- OpenMP
  - Compiler directives
  - #pragma omp construct clause
  - #pragma omp parallel for
Message Based Parallel Programming

- Send/Receive messages to share values
- Synchronous communication
- Asynchronous communication
- Blocking
- Non-blocking
Divide and Conquer

• Divide and conquer algorithms can often be parallelized

• Basic pattern
  – Break problem into 2 halves
  – Execute each half in parallel
  – Combine the results into a final solution

• Example Problem
  – Find the largest value in an array
Fork/Join

- Part of java.util.concurrent
- Available since Java 7
- Manages a thread pool
- Fork creates a new **task** which will be executed by a **thread**
- Two types of tasks
  - RecursiveAction and RecursiveTask
- Join blocks until a task is completed
- If the task is a RecursiveTask, join returns a value
- Created primarily to support divide/conquer algorithms
import java.util.concurrent.;
import java.util.;;

public class Max extends RecursiveTask<Integer> {

    int threshold = 1000;

    int low;
    int high;
    int[] numbers;

    Max(int[] nums, int low, int high) {
        numbers = nums;
        this.low = low;
        this.high = high;
    }
}
public Integer compute() {
    if (high-low < threshold) {
        int max = numbers[low];
        for (int i = low+1; i < high; i++)
            if (numbers[i] > max)
                max = numbers[i];
        return max;
    }
}
int mid = (high + low)/2;
Max left = new Max(numbers, low, mid);
Max right = new Max(numbers, mid, high);
left.fork();
int maxR = right.compute();
int maxL = left.join();
if (maxR > maxL)
    return maxR;
else
    return maxL;
public static void main(String args[]) {
    int count = new Integer(args[0]);
    Random r = new Random();
    int numbers[] = new int[count];
    for (int i = 0; i < count; i++)
        numbers[i] = r.nextInt();
    ForkJoinPool f = new ForkJoinPool();
    int max = f.invoke(new Max(numbers, 0, count));
}